下载APP

12｜Jitter Buffer：拿什么拯救你，花屏卡顿问题

2021-12-17 李江

《攻克视频技术》 课程介绍 >

讲述：李江
时长 23:07 大小 21.19M

你好，我是李江。

之前的两节课我们讲述了如何做好带宽预测和码率控制。好的带宽预测算法能够比较准确地预测出实际的网络带宽，而好的码率控制算法能够使得编码输出码率贴合实际网络带宽。这两个算法是视频流能够在各种网络状况下流畅播放的最基础的前提。

但是在实际情况中，很多时候我们还是会遇到各种各样的卡顿和花屏的问题。相信这两种问题你也是经常遇到，同时也是让你很头痛的问题。

那么，我们这节课就来讨论一下，一般哪些环节出问题会比较容易出现卡顿，以及哪些环节有问题会比较容易出现花屏。并且，我也会给出一些常用的解决方案。通过这节课的学

习，你之后再遇到花屏卡顿问题的时候，可以参考一下，从而能够快速发现问题和找到解决的方法。

在讲述具体问题之前，先简单介绍一下 Jitter Buffer 这个模块。它是好几个卡顿和花屏问题的处理模块。Jitter Buffer 工作在接收端，主要功能就是在接收端收到包之后进行组帧，并判断帧的完整性、可解码性、发送丢包重传请求、发送关键帧请求以及估算网络抖动的。其中组帧、判断帧完整性、判断帧可解码性、丢包重传、关键帧请求都是这节课的重点。因此，我先在这里重点说一下 Jitter Buffer 这个模块。Jitter Buffer 在接收端所处的位置如下图所示：

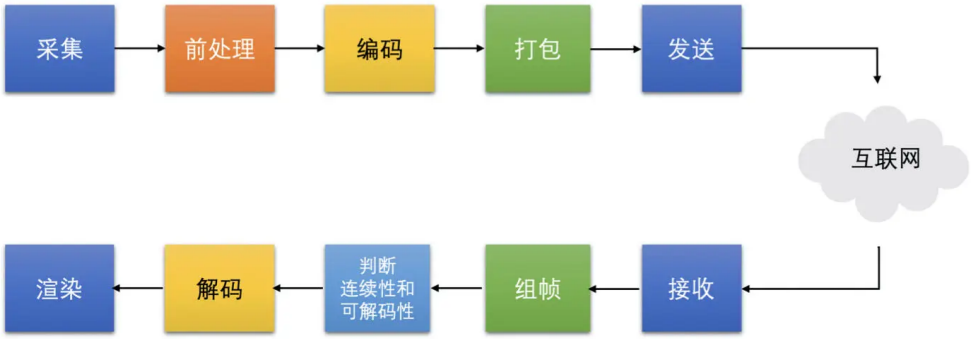
接收端

极客时间

卡顿问题

视频卡顿是在实时视频通话场景中非常重要的一个问题。卡顿率也是实时通话场景中一个非常重要的指标。一般来说，人眼在帧率达到 10fps 并且均匀播放时就不太能看出来卡顿了。**如果两帧之间的播放时间间隔超过了 200ms，人眼就可以明显看出卡顿了。**那一般什么情况下会导致两帧之间的播放时间间隔超过 200ms 呢？我们下面一个个来分析一下，并给出相应的解决方法。

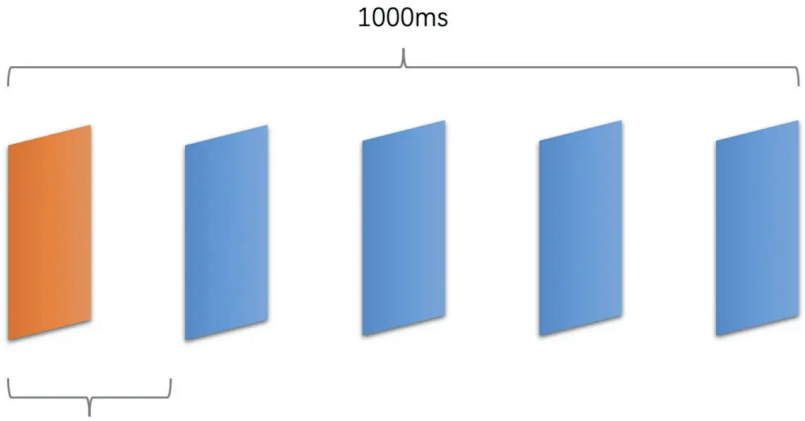
接下来我们按照下图中采集到渲染这条链路中每一个可能引起卡顿问题的原因依次来介绍。



极客时间

帧率不够

如果实际采集到的帧率或者设置的帧率本身就只有 5fps，即便是均匀播放，两帧之间的间隔也会达到 200ms，那么这种情况下肯定会出现卡顿。画面看起来就像是快速播放的 PPT。这种情况下相信你能明显地看出来卡顿的原因。如下图所示：

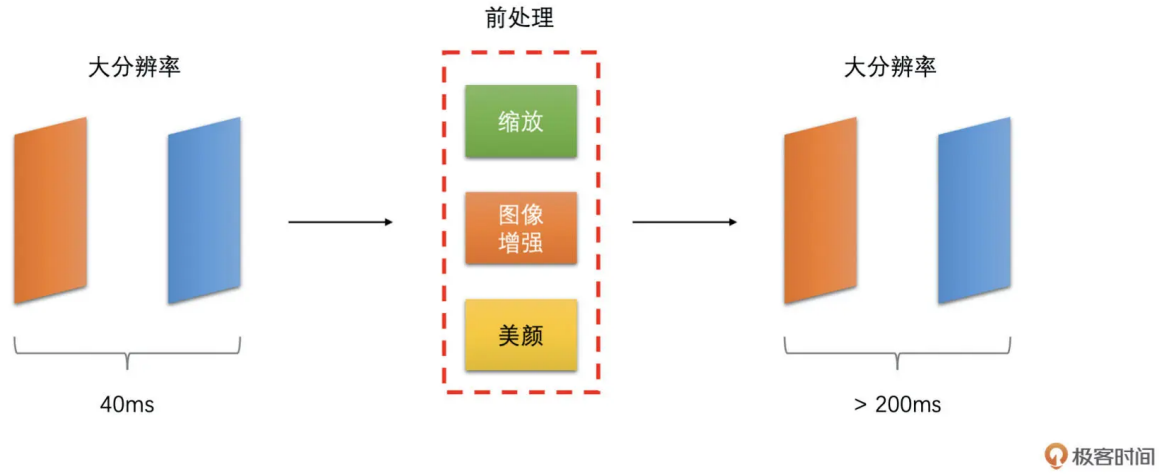


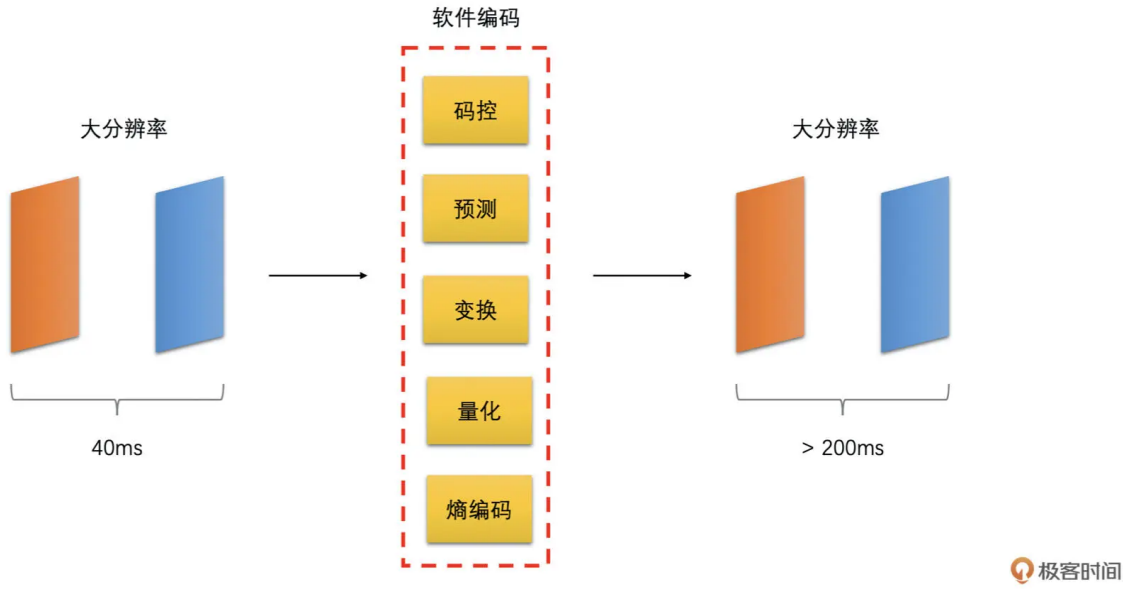
极客时间

这个问题最好的解决方法当然就是提高帧率了，比如提高到 15fps 或者更高。当然有的时候采集帧率就是上不来，那我们就要定位一下，采集帧率不高的具体原因是什么。

机器性能不够，导致前处理或者编码耗时太长

在实时通话场景中，画面是需要实时地做前处理（美颜等操作）并编码之后发送到对端进行解码播放的。如果本身机器性能不够，而画面分辨率又很高，那么这可能会导致前处理一帧或者编码器编码一帧的耗时很高。如下面两张图所示：





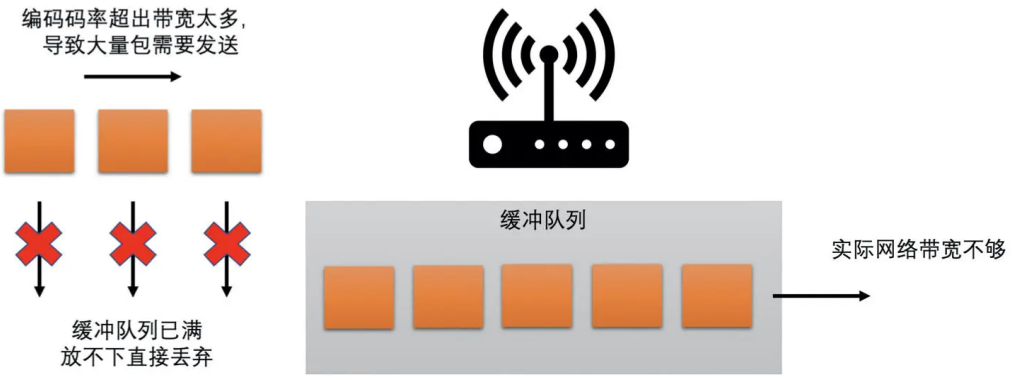
这种情况下，即便是采集的帧率很高，但是前处理和编码操作机器处理不过来，从而最后导致两帧被发送出去的间隔也会很高，这时发送到对端，对端就可能会出现明显的卡顿。**这种情况在比较老的手机上特别容易出现。**

当出现这种情况的时候，我们可以在高分辨率的时候尽量使用 GPU 做前处理，并使用硬件编码或者将软件编码设置为快速档加快处理的速度。GPU 做前处理和硬件编码消耗 CPU 比较小，并且速度更快。软件编码设置为快速档时很多费时间的编码工具都被关闭了，因此可以提高编码的速度。不过这里你需要注意一下，就是这样**压缩率也会下降**。

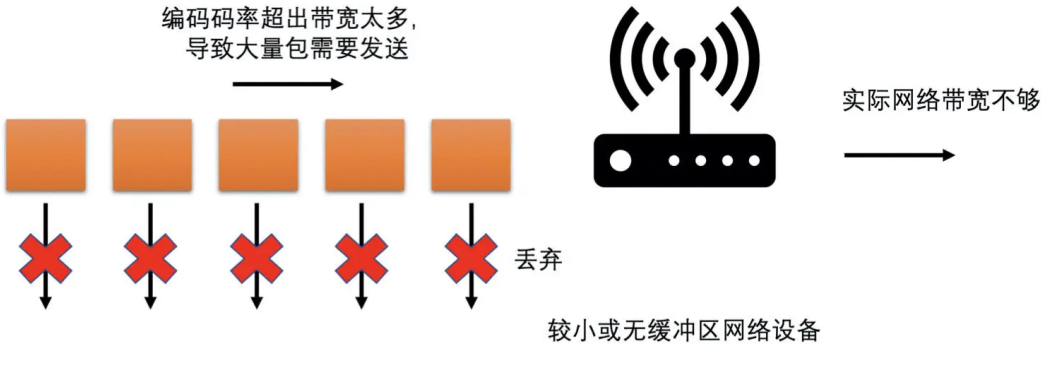
编码器输出码率超过实际网络带宽

这种情况是 RTC 实时通话场景中卡顿问题最常见的根因。当出现的时候往往会引起比较长时间的卡顿，有可能持续 1～2 秒钟时间。有的时候，网络突然变差，从而网络预估出来的带宽很小，但是实际播放的画面很复杂，且需要的编码码率又比较高，这样就比较容易出现发送码率大于实际带宽的问题。

我们在带宽预测的那节课中讲过，当发送码率大于实际带宽的时候，对于有缓冲区的网络设备，它一开始会将包放在缓冲区，且当缓冲区放不下了还是会丢包。



而对于没有缓冲区的网络设备，它是直接就丢包。



当包被丢弃了，对端就不能完整地恢复出一帧图像了。而我们知道，当一帧图像不能解码，那么之后所有参考它的图像就都不能解码了。并且，在 RTC 场景中，我们一般使用连续参考的参考帧结构，就是后面的 P 帧参考它的前一帧，这也就会导致在下一个 IDR 帧到来之前画面都会卡死。这样卡顿的时间就会很长。如果出现这种问题怎么办呢？

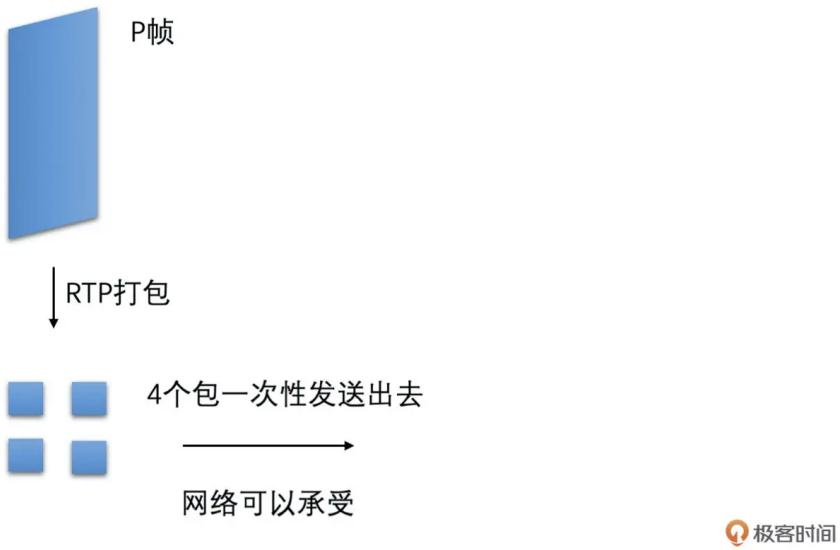
我们需要对发送码率做严格的限制，防止它超过预估带宽。这就需要编码器的输出码率要能够贴合预估带宽。也就是说，我给编码器设置多少编码码率，编码器最好就编码出多少码率。

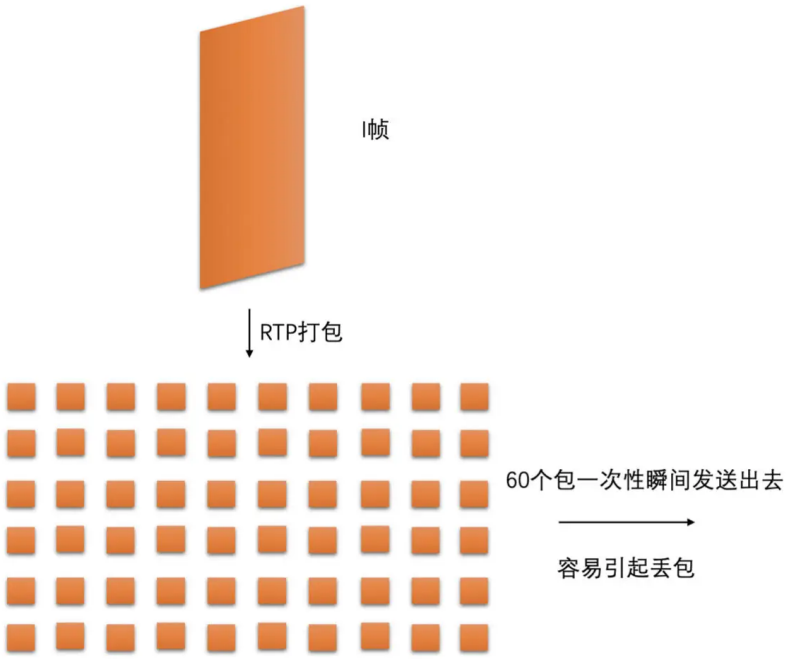
回顾一下我们在第 11 讲中讨论的内容，那是不是我们应该选择 CBR 的码控算法呢？是的，**在 RTC 视频通话场景下我们最好选择 CBR 的码控算法，从而保证输出码率能够比较好地贴合预估带宽。**

如果使用 VBR 码控算法，编码器的输出码率会随着画面的复杂程度变化，那就会有很大的概率因为画面复杂而出现输出码率超过预估带宽的情况，从而导致对端出现严重的卡顿。而 CBR 码控算法是你设置多少目标码率，编码器的输出码率就会接近于目标码率。这样，超发的问题就会少很多，相应地对端出现卡顿的概率也会小很多。

复杂帧编码后过大或者 I 帧比较大

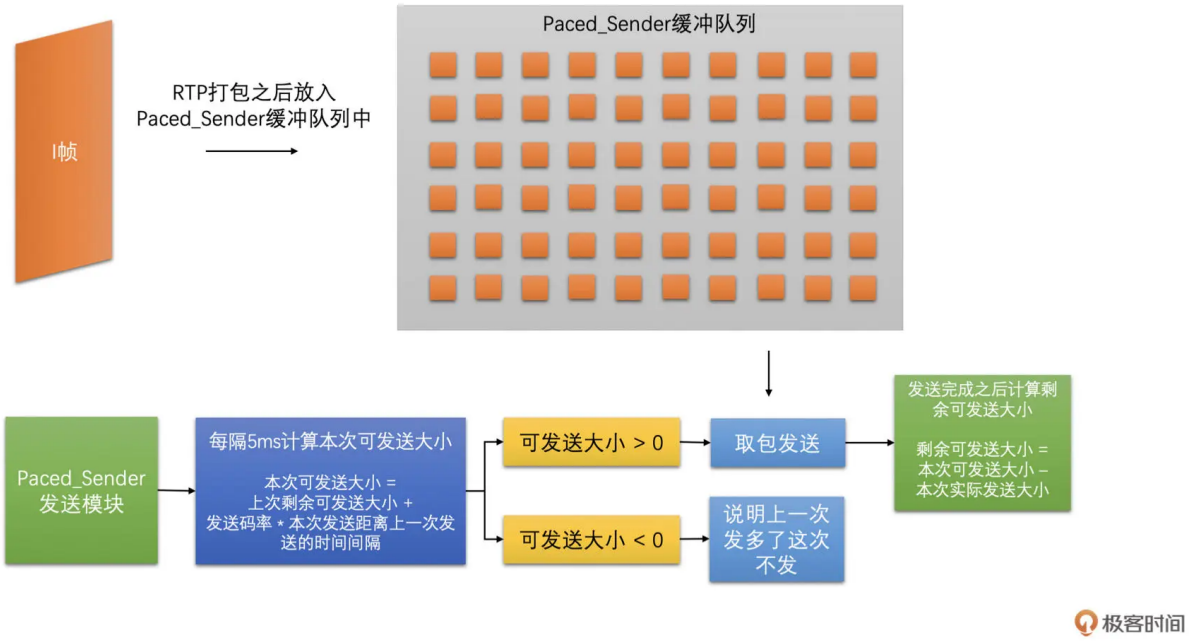
虽然，我们选择使用 CBR 的码控来编码可以使得一段时间内（比如说 500ms 或者 1 秒钟）的编码输出码率尽量地贴合预估带宽，但是有的时候编码画面变化很大的帧或者需要编码 IDR 帧的时候，还是会使得编码后这一帧的大小会比较大。如果一次性将这种大帧打包出来的所有包都直接发送到网络中，则会在一瞬间加剧网络的负担，从而容易引起网络丢包，继而引起卡顿的可能。如下面两张图所示：





为了能够减小这种大帧带来的瞬时网络波动，我们可以在编码打包之后、发送之前，加一个平滑发送的模块来平滑地发送视频包。这个模块在 WebRTC 中叫做 PacedSender（节奏发送器）。那它的工作原理是怎么样的呢？

PacedSender 主要的工作原理就是编码输出的码流打包之后先放到它的缓冲区中，而不是直接发送。之后它再按照预估带宽大小对应的发送速度，将缓冲区中的数据发送到网络当中。如下图所示：



一般 PacedSender 每隔 5ms 左右发送一次包，并且它会在内部记录上一个 5ms 发送周期发送完之后剩余可发送的大小。同时，每隔 5ms 左右，它计算当前距离上一次发送包的时间差，乘上发送码率得到这段时间可以发送多少大小的数据。然后再加上上一次剩余可发送大小得到本次可发送大小。因为发送的时候是一个个 RTP 包发送的，而一般一个包差不多就是 1500 字节，所以上一个剩余可发送的大小可能为负数

如果本次可发送大小大于 0，就从缓冲区中取包发送出去。并且，发送完包之后将剩余可发送大小减去发送包的大小。之后如果剩余可发送大小小于或等于 0，则停止发送，并等待下一个 5ms 发送周期再发送。

PacedSender 是通过控制实际发送码率来平滑发送的，这样能防止编码输出码率超过网络带宽太多，直接将包一次性发送到网络导致卡顿。但是我们要注意，如果编码器输出码率差网络带宽太多，也会导致 PacedSender 缓冲太多数据包，从而引起延时太长。

因此，编码器码控还是需要贴合网络预估带宽的。PacedSender 大多时候是用来防止一两帧编码后太大引起数据量突增造成丢包。因此，**码控和 PacedSender 都很重要，它们是一起协作来减少卡顿的。**

网络本身就有一定的丢包率

当然，我们选择 CBR 的码控同时使用了平滑发送方法，但有的时候网络变化太快了或者我们处在一个无线网络环境下，就是会有一定的丢包概率。那怎么办呢？

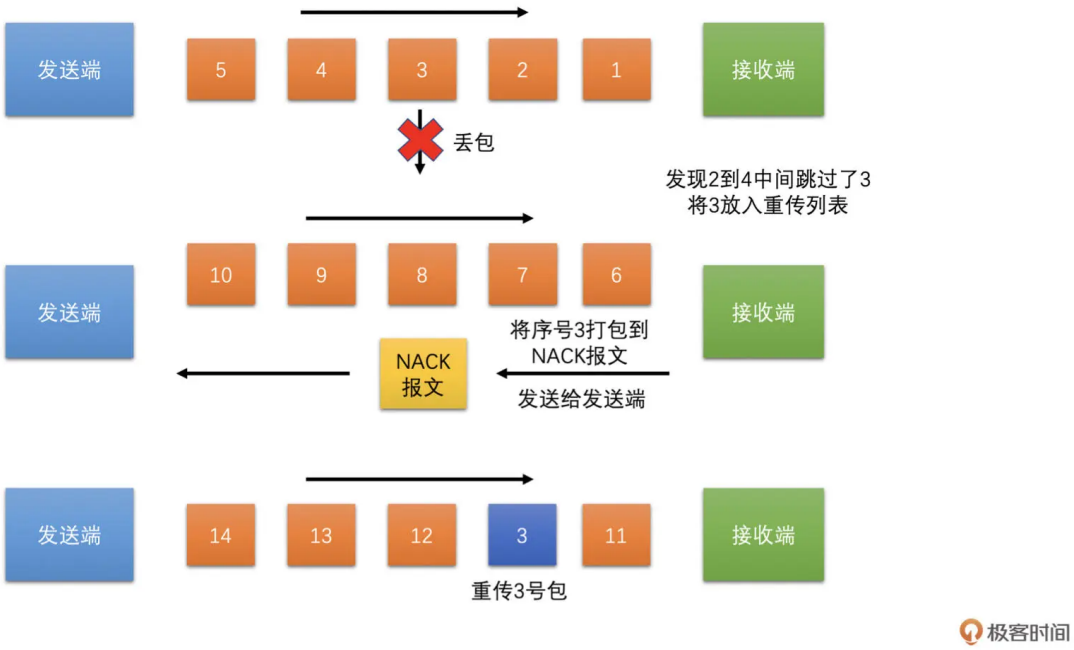
这就要使用我们前面多次讲到过的丢包重传策略了。因为对于视频来说，如果视频帧出现了丢包的话，帧就不完整了，那么当前帧也就不能拿去解码，就可能引起卡顿。如果强行解码，从这一帧开始到下一个 IDR 帧中间的帧，几乎都会出现解码花屏或者解码错误，而解码错误也会引起卡顿。

这个知识点你可以参考第 05～07 这三节课，里面详细讲述了为什么会这样。因此，如果真的出现丢包了，那么我们必须想办法将包恢复。其中，最常用的方法就是丢包重传。

丢包重传请求策略是在 Jitter Buffer 里面实现的。当接收端接收到视频 RTP 包之后，会检查 RTP 序列号。

如果序列号不连续，出现了跳变，也就是说，当前 RTP 包序列号减去收到的最大 RTP 包序列号大于 1，那么就认为中间的包有可能丢失了，Jitter Buffer 就将中间没有收到包的包序号都加入到丢包列表中。因为 UDP 经常会出现乱序到达的情况，如果中间的包后面到来了，也就是说 RTP 包序号小于收到的最大 RTP 包序号，Jitter Buffer 就将这个包序号从丢包列表中删除，防止重复传输。

接收端每隔一定时间将丢包列表组装成 RTCP 协议中的 NACK 报文（我们在第 09 讲中详细讲述过）发送给发送端，并且我们在第 10 讲中也说过，发送端会保存之前的发送历史数据。发送端收到 NACK 报文之后，就会解出 NACK 报文中携带的丢失包的序号，并且在发送的历史数据中找出这个包重新发送给接收端。接收端收到包就将丢包列表中的对应序列号删除。如下图所示：

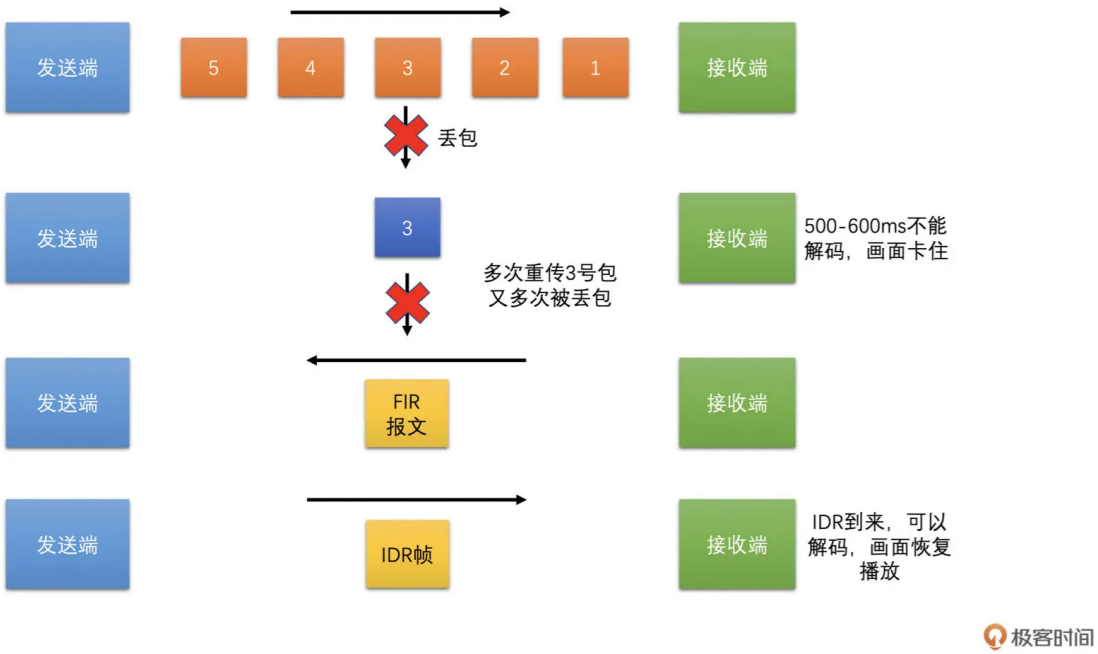


但是这里有一个问题就是，有的包重传一次需要一段时间才能到接收端，因为 NACK 发送给发送端需要时间，重传包传输到接收端也需要时间，中间正好一来一往，差不多一个 RTT（往返时间）时间。

因此，每个丢失包序号发送重传请求之后，下一次需要等一个 RTT 的时间。如果接收端等待一个 RTT 的时间后还没有收到对应序号的 RTP 包，则再次将该序号加入到重传请求中，不能每次 NACK 请求都把所有丢包列表中的序号加入到报文中，防止重传重复发送，加重网络负担。同时重传也是有次数限制的。如果一个包重传请求发送了好几次，比如说 10 次，还没有收到，那就不再将该包加入到 NACK 报文中了。

重传也没有收到包

一般来说，前面的策略用上了之后，卡顿会小很多。但是，有的时候就是会有极端情况出现。毕竟，网络是千变万化的。如果实在是前面策略都用上了，还是出现了有包没有收到，导致帧不完整，继而导致没有帧可以解码成功的话，那么我们就需要使出最后的大招了，那就是关键帧请求，也叫 I 帧请求。**I 帧请求使用 RTCP 协议中的 FIR 报文。这个策略也是工作在 Jitter Buffer 中的。**具体如下图所示：



前面课中我们讲到过，如果有一个帧解码失败，那之后的帧几乎都将解码失败，直到下一个 IDR 帧到来。因此，如果有一帧出现了丢包的情况，导致后面的帧都无法解码了，那么接收端这个时候就需要发起一个关键帧请求报文给发送端，发送端收到关键帧请求之后应该立即编码一个 IDR 帧。这样接收端收到 IDR 帧之后就可以解码了，而前面不能解码的帧就全部删除掉。同时，将丢包列表清空掉。

花屏问题

除了卡顿问题，另外一个比较让人头痛的问题就是花屏问题。花屏问题的出现主要有以下几种比较常见的原因。

帧不完整

前面我们说过，如果帧出现了丢包就送去解码的话，若能解码成功，那肯定会出现解码花屏的问题。尤其是 ffmpeg 作为解码器的时候，帧不完整也有很大的概率成功解码，但是得到解码后的图像却是花屏的。如下图所示：

极客时间

因此，**我们在解码一帧数据之前一定要保证帧是完整的。**那怎么保证呢？

在 RTP 包里面，RTP 头有一个标志位 M，表示是一帧的结尾。因此只要收到这个标志位为 1 的包就代表收到了这一帧的最后一个包。那么如何判断一帧的第一个包有没有收到呢？如果收到了一帧的第一个包，也收到了这一帧最后一个包，那我们就有了一帧的第一个和最后一个包的 RTP 序列号了。只要中间的序列号对应的 RTP 包都收到了，那么当前帧就完整了，是不是？

于是，现在的重点是怎么确定一帧第一个包有没有收到。还记得我们在第 09 讲里面 RTP 打包的时候讲到的：

> 在单包模式打包方式的时候，一帧只打一个包，最后一个包就是第一个包。那只要收到最后一个包就等于收到了第一个包，很容易判断，是不是？
>
> 在组合打包方式的时候，一个包里面有好几个帧，那么，只要按照协议将几帧分离开来就可以。这时帧肯定是完整的。因为包丢了的话，这几帧就都丢了，不存在丢掉帧中的一部分，是不是？

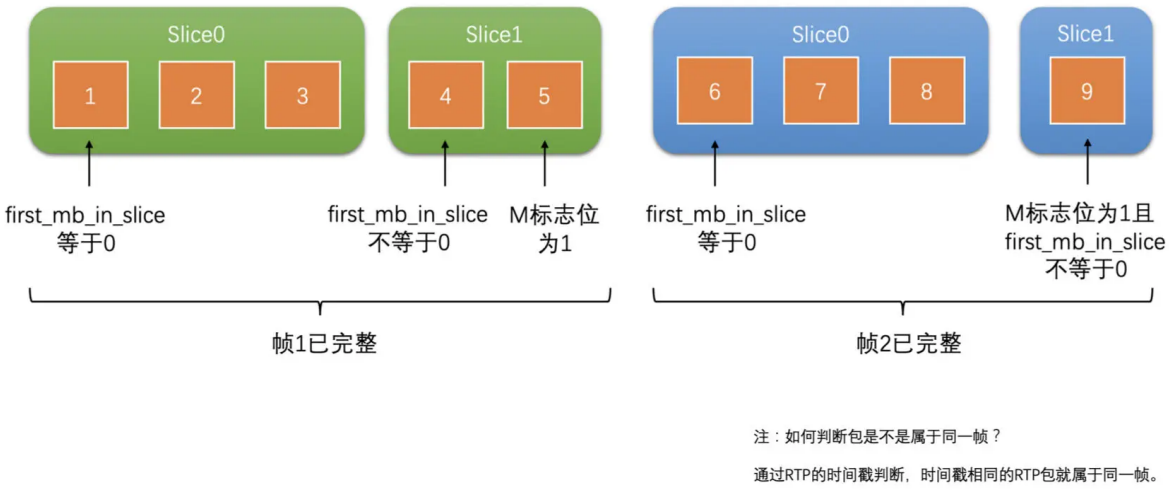
> 问题在于在分片打包的时候，分片打包一帧会分成好几个包打包，而丢掉了一个包，帧就不完整了。我们知道分片打包时 FU Header 里面有一个 S 标志位。这个标志位表示的是第一个包，那我们是不是就可以使用这个标志位来判断是不是收到了第一个包了呢？
>
> 答案是：不是！不是！不是！如果使用这种方法来判断完整性的话，那大概率会出现花屏问题。

这个地方千万要注意，**我们在 RTP 打包的时候是以 Slice 为单位打包的，而不是以帧为单位打包的。**因此，前面几种方式都是只能表示一个 Slice 完整了，而不能表示一帧完整了。因为一帧是有可能有多个 Slice 的。

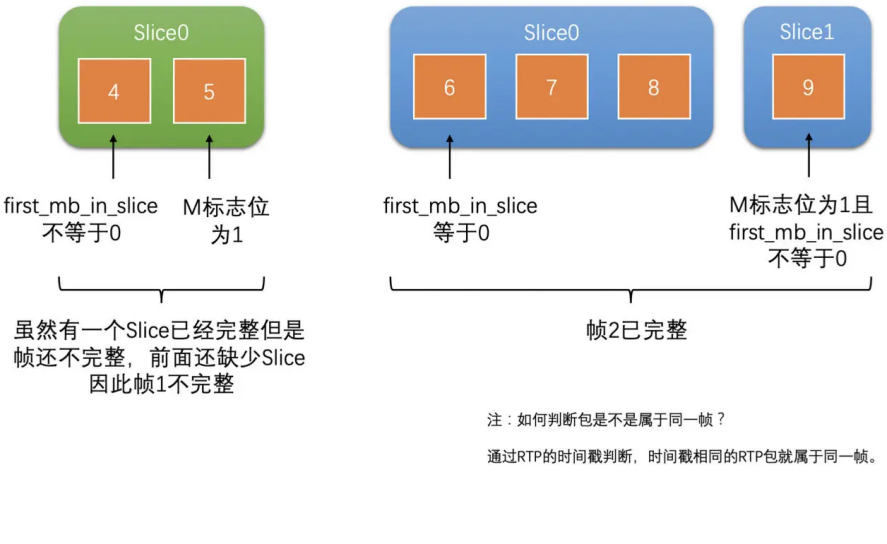
再一次强调前面关于单包、组合打包、分片打包的帧完整性判断都是错误的。**那正确的帧完整性判断应该怎么做呢？**

我们也是在 Jitter Buffer 中来对帧进行完整性判断的。首先，我们使用前面的方式判断 Slice 的完整性，保证一个个 Slice 是完整的。然后，在第 05 讲中我们讲到过使用 slice_header 中的 first_mb_in_slice 字段，来判断当前 Slice 是不是第一个 Slice。如果这个 first_mb_in_slice 字段为 0，就代表是帧的第一个 Slice 了。

我们找到帧的第一个 Slice，而 Slice 也判断了是完整的，再通过 RTP 头的 M 标志位判断了帧的最后一个包。如果第一个 Slice 的第一个包到帧的最后一个包之间的 RTP 包都收到了，那就代表帧完整了。这是一种方法。如下图所示：

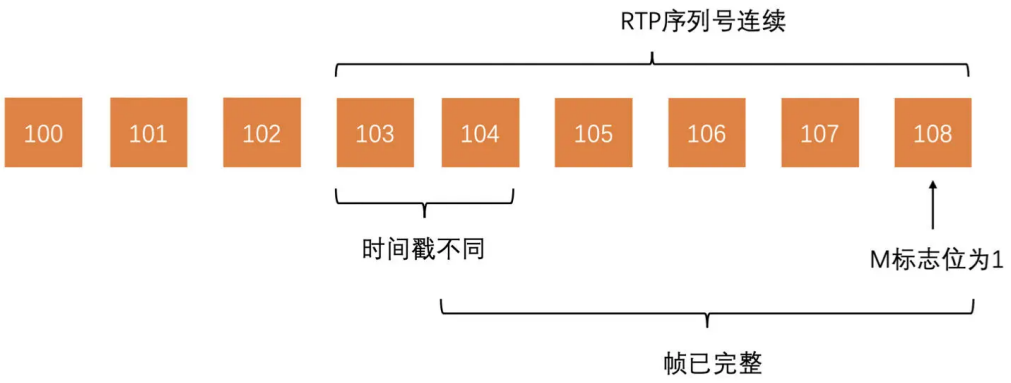


从上面的图中我们可以看到两个帧都是完整的。而下面图中的帧 1 是不完整的。



还有另外一种方法，是 WebRTC 中在使用的方法，就是将每一个收到的包都排好序放在队列里面。Jitter Buffer 收到了当前帧的最后一个包（RTP 头的标志位 M 为 1）之后呢，从这个包往前遍历，要求 RTP 序列号一直连续，直到 RTP 时间戳出现一个跳变，代表已经到了前面帧的包了。

如果包序号一直是连续的，那么代表当前帧就是完整的了。因为两帧的时间戳是不可能一样的。这也是一种方法。但是这种方法需要所有包都放在一个队列里面，并且排好序。它没有前一种方法灵活。如下图所示：



注：如何判断包是不是属于同一帧？

通过RTP的时间戳判断，时间戳相同的RTP包就属于同一帧。

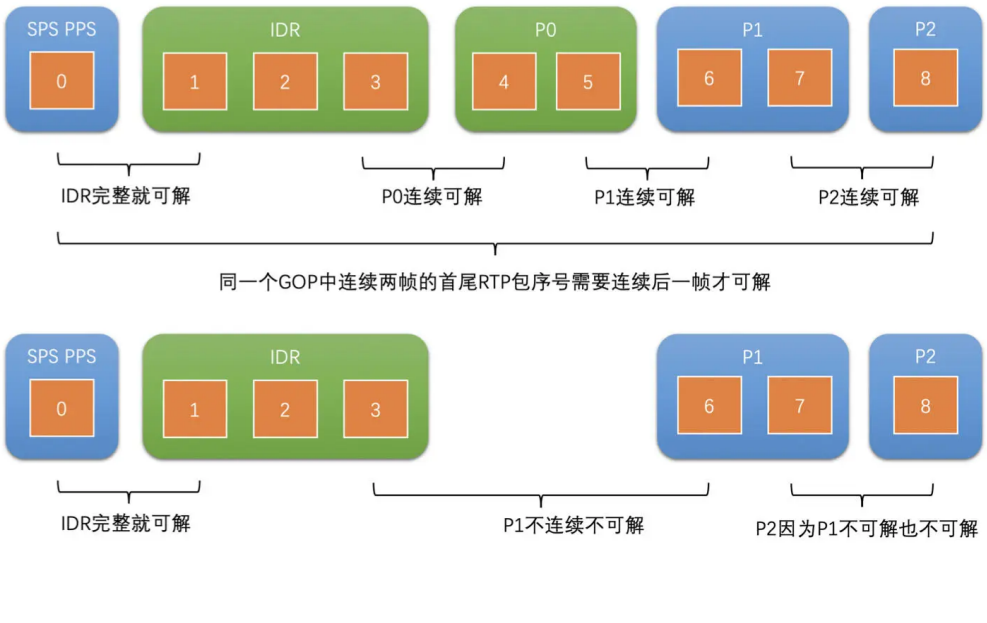
极客时间

参考帧不完整

当一帧完整了之后，我们是不是就可以将帧送去解码，就不会出现花屏了呢？答案是：不是这样的。

因为，我们前面强调过很多遍，需要参考帧也是完整的才能送解码，并且参考帧的参考帧也要是完整的才行。如果参考帧不完整或者丢失，会出现如下图所示的花屏。

那也就是说，如果是连续参考的话，或者说你不知道编码器使用的参考结构的话，你就需要保证从 IDR 帧开始到当前帧为止所有的帧都是完整的，并且前面的帧都已经解码了，那当前帧才能送去解码。因为只要有一帧没有解码就会出现花屏。具体如下图所示。这部分功能一般也是实现在 Jitter Buffer 中。



当然，有的时候我们并不一定使用连续参考，比如，我们下一节课会讲到的 SVC 编码，就不是连续参考的。那就不要求前面的帧都完整才可以解码。或者，你自己设计了参考结

构，并不是使用了连续参考的方法做编码的，也不需要要求所有的帧都完整。

这个时候你需要设计自己的协议告诉接收端什么时候是可解码的，防止出现花屏。如果没有的话，那就当作连续参考处理，防止花屏的出现。因为这种情况花屏一旦出现，到下一个 IDR 帧到来都会一直花屏，这是不能接受的。

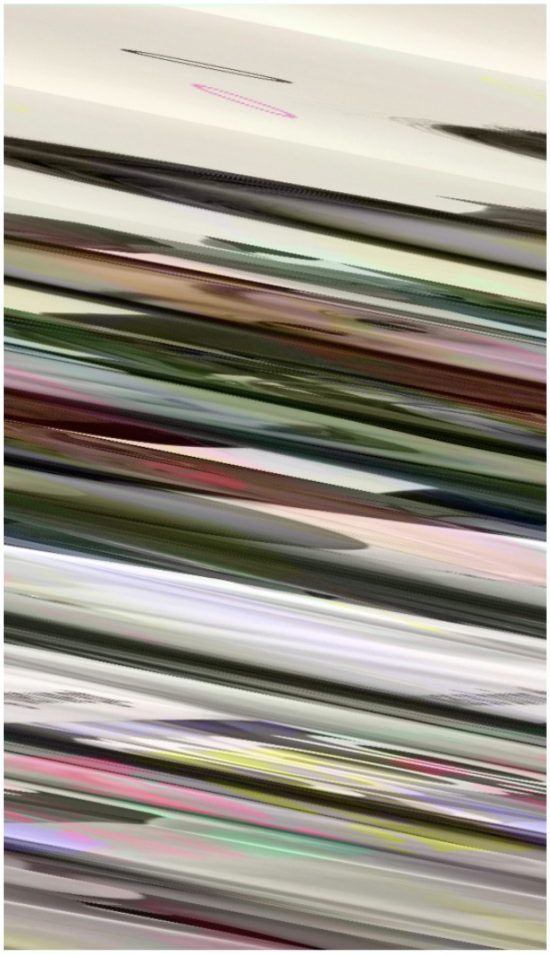
YUV 格式问题

另外一种常见的花屏问题，就是**渲染的时候 YUV 格式弄错了**。这种问题经常会出现，我们声网的客户就出现了好几次没有处理好这个问题导致的花屏。这种情况有一个特点，就是图像的大体轮廓是对的，但是颜色是有问题的。如下图所示，左图 YUV 格式是正确的，而右图 YUV 格式是错误的。

根因是 YUV 中的 Y 分量是对的，但是 UV 是错误的。这种时候你就应该想到，很有可能就是 NV12 当作 NV21 处理了，或者 I420 当作 NV21 处理了，类似这种 YUV 格式弄错了的问题。其处理方式也很简单，就是使用正确的 YUV 格式就对了。渲染或者读取 YUV 的时候一定不要弄错了 YUV 的类型。

Stride 问题

最后一种花屏问题，是老生常谈的问题啦。那就是 Stride 问题。**解码后渲染前一定要处理好 YUV 的 Stride 问题，不要和宽度弄混了。**如果出现类似下图的现象的话，去看看你的 Stride 是不是弄错了吧。



小结

好了，今天的课到这里就要结束了。我们来回顾一下这节课的主要内容。

我们今天主要介绍了哪些环节出现问题会比较容易引起卡顿和花屏，并给出了相应的解决方法。

1. **卡顿**

   在采集或设置的帧率不够时，会造成两帧之间的时间间隔过长，从而导致卡顿。我们需要提高帧率。

   机器性能不够，导致前处理或者编码耗时太长，从而导致卡顿。我们可以在高分辨率的时候尽量使用 GPU 做前处理，并使用硬件编码或者将软件编码设置为快速档加快处理的速度。

   编码器输出码率超过实际网络带宽。在 RTC 视频通话场景下我们最好选择 CBR 的码控算法，从而保证输出码率能够比较好地贴合预估带宽。

复杂帧编码后过大或者 I 帧比较大，如果一次性将这种大帧打包出来的所有包都直接发送到网络中，容易引起网络丢包，继而引起卡顿的可能。为了能够减小这种大帧带来的瞬时网络波动，我们可以在编码打包之后、发送之前，加一个平滑发送的模块来平滑地发送视频包。这个模块在 WebRTC 中叫做 PacedSender。它的主要工作原理就是编码输出的码流打包之后先放到它的缓冲区中，而不是直接发送。

网络本身就有一定的丢包率，如果视频帧出现了丢包的话，帧就不完整了，那么当前帧也就不能拿去解码，可能引起卡顿。最常用的方法就是丢包重传。丢包重传请求策略是在 Jitter Buffer 里面实现的。

重传也没有收到包，导致帧不完整，继而导致没有帧可以解码成功，我们就需要使用关键帧请求，也叫 I 帧请求。I 帧请求使用 RTCP 协议中的 FIR 报文。这个策略也是工作在 Jitter Buffer 中的。

2. **花屏**

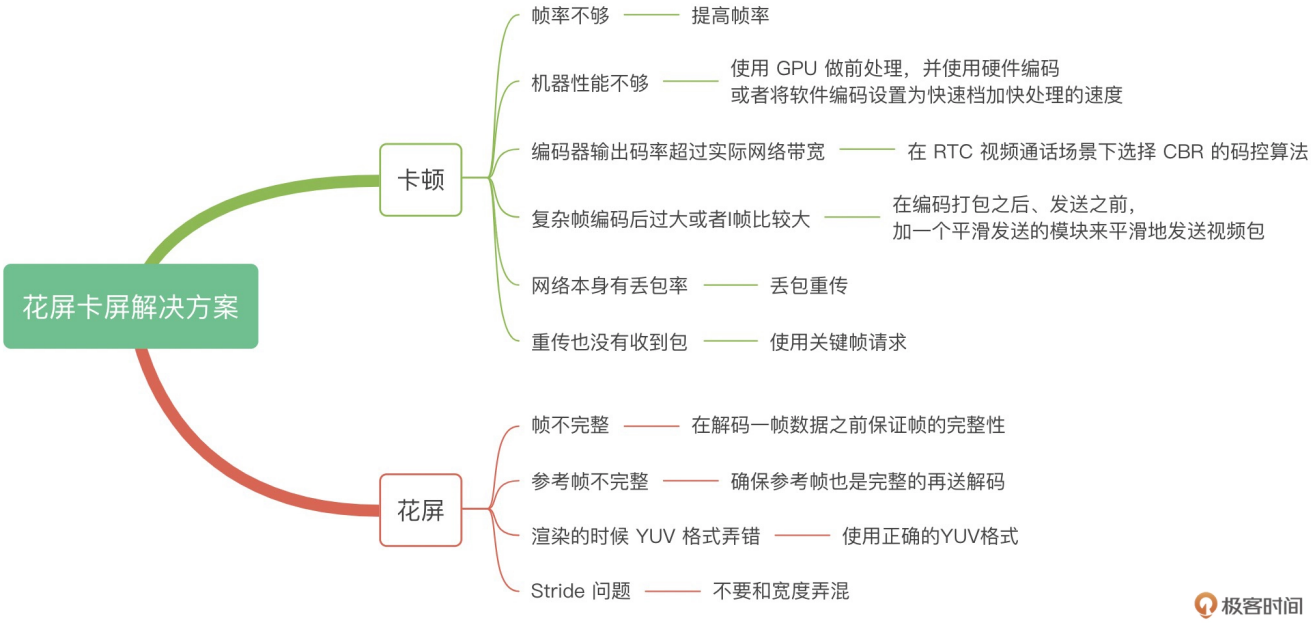
帧不完整。如果帧出现了丢包就送去解码的话，若能解码成功，那肯定会出现解码花屏的问题。因此，我们在解码一帧数据之前一定要保证帧是完整的。记住我们在 RTP 打包的时候是以 Slice 为单位打包的，而不是以帧为单位打包的。我们也是在 Jitter Buffer 中来对帧进行完整性判断的。

参考帧不完整。我们需要参考帧也是完整的才能送解码。如果参考帧不完整或者丢失，会出现花屏。这部分功能一般也是实现在 Jitter Buffer 中。

渲染的时候 YUV 格式弄错了，也会造成花屏。其处理方式也很简单，就是使用正确的 YUV 格式就对了。

Stride 问题会造成花屏。我们解码后渲染前一定要处理好 YUV 的 Stride 问题，不要和宽度弄混了。

为了方便你记忆，这里我给出了一张总结图。



思考题

今天我们的思考题换一种形式，你来说说你遇到过哪些引起卡顿和花屏问题的原因吧。我们一起在评论区相互学习交流解决这类问题的经验。

你可以把你的问题和疑惑写下来，分享到留言区，与我一起讨论。我们下节课再见。

分享给需要的人，Ta订阅后你可得 20 元现金奖励

生成海报并分享

赞 0 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 11 | 码控算法：如何控制视频的编码码率？

精选留言

写留言

由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。